**Simon Schliecker,**
**Mircea Negrean,**
**Rolf Ernst**

# Reliable Performance Analysis of a Multicore Multithreaded System-On-Chip (with appendix)

**Braunschweig : Institute of Computer and Communication Network Engineering, 2008**

Technical Report / Technische Universität Braunschweig ; 22837

Veröffentlicht: 13.10.2008

http://www.digibib.tu-bs.de/?docid=00022837

Auch erschienen in:
International Conference on Hardware-Software Codesign and System Synthesis

# Reliable Performance Analysis of a Multicore Multithreaded System-On-Chip

(with appendix) *

Simon Schliecker, Mircea Negrean, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technische Universität Braunschweig, Germany
schliecker@ida.ing.tu-bs.de

## ABSTRACT

Formal performance analysis is now regularly applied in the design of distributed embedded systems such as automotive electronics, where it greatly contributes to an improved predictability and platform robustness of complex networked systems. Even though it might be highly beneficial also in MpSoC design, formal performance analysis could not easily be applied so far, because the classical task communication model does not cover processor-memory traffic, which is an integral part of MpSoC timing. Introducing memory accesses as individual transactions under the classical model has shown to be inefficient, and previous approaches work well only under strict orthogonalization of different traffic streams.

Recent research has presented extensions of the classical task model and a corresponding analysis that covers performance implications of shared memory traffic. In this paper we present a multithreaded multiprocessors platform and multimedia application. We conduct performance analysis using the new analysis options and specifically benchmark the quality of the available approach. Our experiments show that corner case coverage can now be supplied with a very high accuracy, allowing to quickly investigate architectural alternatives.

## 1. INTRODUCTION AND MOTIVATION

Formal performance analysis is regularly applied in the design of distributed embedded systems. There, it greatly contributes to an improved predictability and platform robustness of highly complex networked systems, such as in automotive electronics. Advances in new modular performance analysis techniques allow to analyze large scale, heterogeneous systems, providing reliable data on transitional load situations, end-to-end timing, memory usage, or packet losses. The corresponding methods and tools are now regularly used i.e. in automotive design at early industrial adopters [1]. There, analysis is often combined with tracing and simulation to cover the difficult corners of the system state space resulting from parallel execution in distributed applications and communication over heterogeneous networks. Formal analysis is also used for early evaluation of architectures with respect to extensibility or flexibility in combination with design space exploration support.

Improving predictability is also a major goal in MpSoC design in order to reduce design risk and avoid performance bottlenecks. Predicting the timing behavior of MpSoCs, however, is fundamentally more difficult than in the distributed case: The interaction and correlation between integrated system components, such as a shared memory or coprocessors, or cache accesses are highly dynamic and can routinely lead to overload situations. In the application that we present in this paper, memory transactions of 4 multithreaded cores are tightly interleaved partially hiding each other's memory transaction delays.

In this paper, we combine two techniques to solve the performance analysis challenge of a realistic multicore multithreaded system. The first technique addresses performance data acquisition and is adopted from automotive design experience. Rather than pursuing guaranteed worst case execution time analysis, the individual components (i.e. tasks mapped to cores) are simulated individually leading to observed worst case timing and memory access frequencies including caches misses. Other than in the case of distributed real-time systems however, the exact timing of memory accesses of a core triggered by the tasks running on the core, shows large time variations due to architecture and fine grain task behavior. It is therefore hardly possible to derive a guaranteed sequence and timing of such events from measurements.

In previous work, it has been proposed to aggregate the memory accesses over the task execution time and derive event models from such aggregate behavior rather than looking at individual memory transactions. This perfectly matches the data acquisition by measurement. Therefore, we employ the corresponding analysis as a second technique in the multiprocessor analysis. This combination is done for the first time.

Using this procedure we tackle three major obstacles that have hindered the general application of formal methods in the performance analysis of MpSoCs and the given setup in particular: We tackle the timing feedback of memory access on the task execution by explicitly modeling memory accesses; we extract the dynamic run-time behavior of the

*This work is an extended version of: S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable Performance Analysis of a Multicore Multithreaded System-On-Chip, *in Proc. CODES+ISSS*, ©ACM 978-1-60558-470-6/08/10, 2008.


Technical Report 22837, Technische Universität Braunschweig, Germany,

tasks by simulating each task in isolation, thus avoiding system level distortions; and we avoid the inaccuracy of treating memory access individually by resorting to the aggregate modeling.

The results are compared to other performance verification techniques. A simulation of the whole multicore system with the same application load provides a lower bound on the resulting global system performance. We also present a formal verification on the basis of individual transactions which provides an upper bound and represents the previous state of the art in formal performance modeling.

The remainder of this paper is structured as follows. First, we present related work in Section 1.1. We then present the investigated platform and application in Section 2. This is followed by a description of the utilized formal analysis procedure in Section 3. Section 4 provides the results of the experimental application, and we conclude in Section 5.

## 1.1 Related Work

Most previous work has addressed the MpSoC analysis challenges only in part. For example, to avoid the feedback effect of memory timing on the task execution, an increasingly common counter-measure is the orthogonalization of system resources [2][3], e.g. through time-driven scheduling of the memory bus. By reducing the timing interdependence, system functions can then be verified separately. While this option simplifies the verification procedure, it implies a conservative design with in general increased resource and possibly also power requirements. Andrej et al [4] have significantly reduced the cost of orthogonalization by deriving optimal bus-schedules given the memory access pattern of each task. Still, if the same performance can be achieved (and verified) without such hardware mechanisms this allows constructing more efficient and flexible systems.

Isolated task worst case execution time analysis has until recently focused on the single-processor case (see [5] for an overview). As memory access timing is relatively predictable in such a setup, the problem of deriving the memory access delay was for many years simply a matter of deriving the amount of memory accesses (i.e. cache misses) per task execution. Due to the challenges of formally addressing single-processor architectures (out-of-order pipelines, conditional execution, a.s.o.), simulation and measurements are still a common option to derive the relevant information about individual tasks [1, 6]. If these metrics are deemed unreliable, these values may be manually modified to compensate for anticipated and unanticipated changes during the design process. Siebenborn et al. [7] integrate inter-task communication into the control-flow graph representation to cover the globally possible execution traces, covering synchronization effects but not implicit memory delays.

Finally, memory accesses typically occur in great numbers, while real-time research has classically focussed on the individual worst-case. To address this various methods have been suggested. Stohr et al. [6] suggest a simulation-based approach to derive the timing parameters of the arbitration points. They are able to approach PC-like architectures but do not address the overestimation given above. Schliecker et al. and Henriksson et al. [8] have identified the need to investigate the aggregate delay over all memory accesses. Henriksson provides extensions of network calculus to derive the heterogeneous memory access delays. However, they do not consider local scheduling or fully the feedback effect when additional delays may occur due to the stretched execution of a task. This has been addressed in [9] where the aggregate memory access time is derived iteratively.

In multiprocessor systems with shared interconnects and memories formal models can provide insight into worst-case access delays to shared resources. But previous work has provided only insufficient experimental data to demonstrate its applicability to actual real-time systems. For this reason, this paper investigates an industrial-grade MpSoC platform with multithreaded processors. We apply the only system-level approach that allows to address dynamic memory scheduling and its effects on local scheduling. In order to focus on the effects of integrating multiple applications into the same system, we chose simulation as the most efficient method to derive the timing of individual tasks.

# 2. THE STEPNP PLATFORM

The StepNP platform [3] has been introduced the STMicroelectronics advanced system technology organization as an experimental MpSoC target platform for the MultiFlex platform mapping tools [10]. It is general-purpose, but can be adopted to suit the demands of various application domains. StepNP is not used in a commercial product, but it has served as a baseline to support the exploration of platform mapping tools for next generation platforms (such as Nomadik(tm) [11]) The StepNP platform is still very interesting for investigation, as it represents a realistic system. A number of applications have already been ported to the platform [12][10] to investigate application behavior and tune architecture design decisions.

## 2.1 Platform Architecture

The basic StepNP platform consists of a set of fully programmable RISC processors and a standardized interconnect. Figure 1 shows the three basic components of the platform: processor engines (in this case 4 RISC based processors), an interconnect (the STBus communication infrastructure), and some specialized coprocessors (in this case, two hardware-based scheduling engines which support SMP and message-passing programming models [10]).

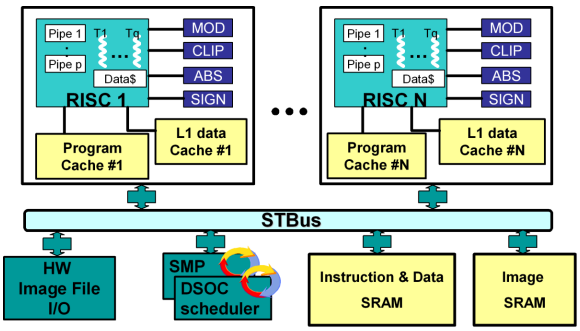


**Figure 1: StepNP Base Platform**

As sketched in the introduction, memory access latency is an issue of growing concern in any embedded system design. In the given platform concept the processors are therefore equipped with hardware multithreading capability. It allows effective latency "hiding" where CPU cycles are not wasted but can be used by other threads. Such a hardware multithreaded processor has a separate register bank

for each thread, allowing low-overhead context switching between threads, often with no disruption to the processor pipeline [3].

In the original concept, a crossbar and a multi-bank memory are used to deliver orthogonal performance to each processor. In this paper, we utilize reliable load models to bound the impact of shared interconnect on timing. One result is, that in the example application, a single shared bus would also deliver sufficient performance.

## 2.2 Image Processing Application

The example application chosen for this investigation was was selected and provided by the École Polytechnique de Montréal and has been mapped to the StepNP platform. It is an image processing algorithm for video applications that consists of 5 successive filtering and processing steps (see Figure 2). Each of these 5 application functions fetches the resulting image produced by the predecessor from the cache or implicitly from the shared memory, performs its necessary operations (mostly on the cached data), and leaves the result in the shared memory for the next stage. The frames are processed sequentially. Each processing step can be parallelised into $n = 2^x$ independent tasks, where $x$ is configurable. The parallelization represents a spacial dissection of the original frame into equally sized tiles. When a new frame has arrived at the system's input the task is forked into $n$ subtasks that are assigned to the available threads. After all subtasks have completed execution the image is merged again for the next step.

For efficiency reasons, no software multiplexing is implemented, so that the number of forked threads is bounded by the number of available hardware threads (number of CPUs multiplied with the number of threads per CPU). The forking and merging is controlled by a user thread running on one of the CPUs in between the pipeline functions. All memory operations pass via the same interconnect to the same memory (see Sec. 2.1).

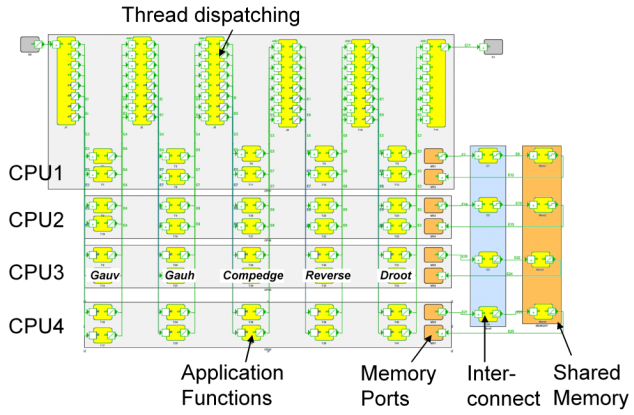


**Figure 2: Image Processing Application**

# 3. FORMAL MULTIPROCESSOR PERFORMANCE ANALYSIS

The traditional approach to formal performance analysis is performed bottom up: First the individual task behavior is investigated in detail to gather all relevant data such as the execution time. This information can then be used to derive the behavior within loosely coupled components, accounting for local scheduling interference. Finally, the system level timing is derived on the basis of the lower level results. This procedure is summarized in this section.

To tackle the analysis complexity of large-scale and heterogeneous systems, the performance analysis can be broken down into separate local analyses of tasks mapped to resources that are then composed using a generic description of the traffic that can lead to task activations (as is done in [13] and [14]).

In general, a task can be a computation, communication, or data storage operation. A task is assumed to be activated when it has all data required for execution available at its inputs. After it has executed for a time no longer than its worst case execution time (WCET), it has produced all data at the output when it has finished. This model of a task corresponds to common design practice in distributed systems. Implicit memory accesses can be covered by the extension described below.

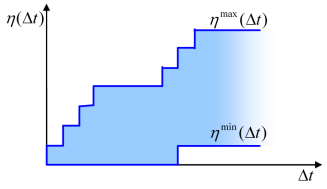


**Figure 3: Example Event Arrival Bounds**

Event models are used to capture the possible patterns of task-activating events in a systematic, abstract fashion. The event models specify the minimum ($\eta^{min}(w)$) and maximum ($\eta^{max}(w)$) amount of events in a stream that may occur in a time window of any given size $w$. To describe the pattern of events in a compact fashion, event models can also be represented through key parameters (such as period, jitter, minimum distance) as is done in [13]. Figure 3 shows the upper and lower bounds of an example (bursty) event model.

Every task is mapped to a resource that defines the scheduling policy used to arbitrate between multiple active tasks. A scheduling analysis (such as those derived from the fundamental work in [15]) can be performed for each resource if the pattern of activating events is known. The result of this analysis is the local task worst case response time (WCRT). Based on this the pattern of activating events that is produced at the task output (which can be system output or another task's input) can be derived (e.g. by accounting for an increased jitter).

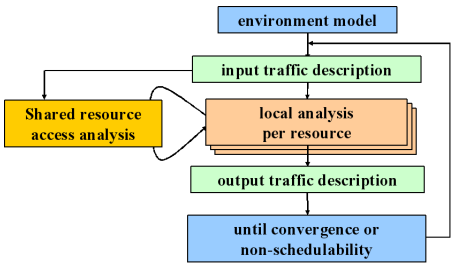


**Figure 4: MpSoC Performance Analysis Loop**

To derive the actual system performance, an iterative approach is used (shown in the outer loop at the right hand side of Figure 4). First, the traffic imposed onto the system from outside is characterized by the designer in the form of conservative event models. All other event models within the system are initialized with optimistic guesses. These event models are then used as the basis for the local component scheduling analyses as described above. This provides local response times and generated output traffic. These output event models are then used to refine the previous estimates.

This procedure is monotonic, as the event models become increasingly more general with each iteration, and thus each iteration contains the previous assumptions [13]. The analysis is complete if either all event streams converge toward a fix-point, or if an abort condition, e.g. the violation of a timing constraint has been reached. Once the analysis has converged, the local response times can be used to derive end-to-end latencies, and the output event models describe the traffic produced by the system's outputs.

This procedure has been extended in [9] to account for shared memory systems. The model of the task behavior is extended to include local execution and memory transactions during the execution. Such a *communicating task* performs *transactions* during its execution as the ones depicted in Figure 5. The depicted task requires two chunks of data from an external resource. It issues a *request* and may only continue execution after the transaction was e.g. transmitted over the bus, processed on the remote component and transmitted back to the requesting source. Such memory accesses may be explicit data fetch operations or implicit cache misses.

The memory is considered as a separate component and a (local) analysis must be available to predict the timing of a set of memory requests. For this again, the event models capturing the memory traffic are required. Each processor scheduling analysis can then account for memory access timing by calling the memory analysis with locally derived memory event models and additional information (such as addresses). This is shown on the left hand side of Figure 4.

## 3.1 Round-Robin Scheduler

Single-processor round-robin scheduling has been covered by previous research, most recently in [16]. The scheduler provided in the StepNP hardware mulithreaded processor models used in this given case is different mainly in two ways: Firstly, all time slots are of equal size and execution times are an integer multiple of the time slot size (which can be exploited to derive a more compact analysis), and secondly, tasks that are waiting for external data to arrive are skipped (which needs to be addressed by the analysis). This is covered in [17], but the approach can be only applied to systems with up to two threads for which the analysis already exhibits a high computation time. Furthermore, individual minimum memory access times must be given. The response time analysis in the extended version of this paper [18] specifically covers the given scheduler and will be utilized for our analysis.

Almost all tasks in the application are communicating tasks, as they require data from the shared memory during their execution. The response time of a communicating task is given by the sum of its ready times plus the time it is waiting for data.

Figure 5 shows an example execution trace of a task running on core 0. The thread of task 1 is locally preempted by the other active thread and delayed by its memory accesses ("task 1 waiting"). Its memory accesses in turn are delayed by the memory requests coming from the other cores (not shown), but also from core 0 itself. We call the sum of the waiting times the *accumulated busy time*. When the memory request is finished, the task additionally has to wait until its thread context is serviced again.

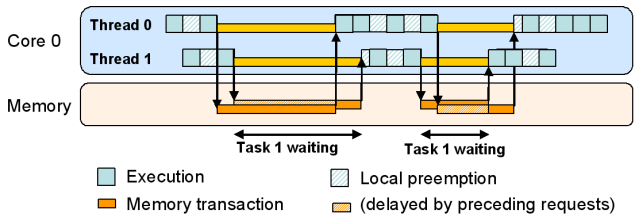


**Figure 5: Example Execution Trace for a Task accessing the Shared Memory**

The challenge in the given scheduling policy is to consider the delay due to memory accesses during the execution of a task. As discussed in the introduction, considering requests individually will lead to a significant overestimation of the actual worst case behavior. The key idea is to consider all requests during the runtime of a task jointly.

This accumulated busy time can be efficiently calculated e.g. for a shared bus: A set of requests is issued from different processors that may interfere with each other. The exact individual request times are unknown and their actual latency is highly dynamic. Extracting detailed timing information (e.g. when a specific cache miss occurs) is virtually impossible, and considering such details in a conservative analysis highly exponential. Consequently, we waive such details and focus on bounding the accumulated busy time.

Without bus access prioritization, it has to be assumed that it is possible for every memory access issued by *any* processor during the runtime of a task activation $i$ that these will disturb the transactions issued by $i$. In the present setup this is given by the requests issued by the other concurrently active tasks on the other processors, as well as the tasks on the same processor as their requests are treated first-come-first-served.

Thus, the accumulated busy time $S$ of a task $\tau_i$'s memory requests can be bounded as follows:

$$S_i(w) \quad \leq \quad \sum_{p \in P} \sum_{\tau \in p} \eta_\tau^+(w) C_\tau \qquad (1)$$

- $P$ is the set of processors in the system. $\tau$ is a task mapped to a processor $p$.
- $\eta_\tau^+$ is the maximum number of requests sent by all activations of task $\tau$ within a time window of size $w$.
- $C_\tau$ is the maximum time that a request by task $\tau$ occupies the shared resource.

The requests of the analysed task $\tau_i$ are considered in Equation 1 as $\eta_i^+(w)$. Please refer to [18] for more detailed modeling, i.e. differentiating $\tau_i$'r requests from the interference by other tasks and options for request prioritization.

Note that the given accumulated busy time depends on the time window size within which the requests are sent. A stretched execution time due to memory accesses allows for additional interference on the memory and vice versa

(increased $\eta_\tau^+(w)$). Thus, Equation 1 needs to be integrated into the tasks response time equation and solved iteratively.

Given a certain dynamism in the system, this accumulative approach will interestingly not result in excessive overestimations as demonstrated in the following experiments.

# 4. EXPERIMENTS

In the first experiment, we investigate the performance analysis accuracy using a synthetic example. Consider a platform configuration with 4 cores connected to a shared memory that is arbitrated first-come-first-served. One core executes a real-time task and the others perform latency insensitive image processing. Due to the common memory and interconnect, the computation on each core can not be considered independently. Rather, the current memory load from any of the cores impacts the run-time of tasks on the other cores.

Assuming each processor thread can have only one open transaction at a time, the worst case memory access time can be straight-forwardly bounded as the product of the number of processors and worst-case delay of each access. This time can be multiplied with the amount of memory accesses and added to the task's core execution time. This method is depicted in Figure 6 (Analysis "per access"). If the same system is executed on the simulator, a much smaller response time is measured for the real time task (Simulation). The ca. 100% deviation shows the room for improvement. Repeating the analysis by resorting to the new analysis options, particularly the accumulated busy time (Analysis "accumulate"), delivers much tighter results.

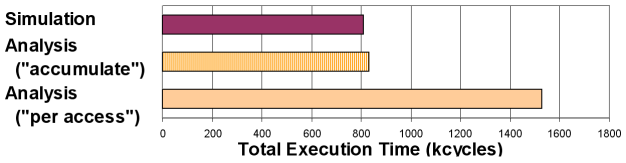


**Figure 6: Formal Analysis options compared to Measurements**

For the following study of the complete system described in Section 2, we adopt a mixed methodology. We use the available timing aware simulators to investigate the timing of individual components (i.e. tasks) in a reasonable amount of time. This removes the need to derive specific models of the tasks and their execution environment. The formal analysis framework presented in this paper is then used to quickly and reliably derive the integration effects on the system level with robust accuracy. Nevertheless, formal methods such as reviewed in [5] can be used to achieve higher confidence in the extracted task timing and consequently the overall analysis results.

We collected the data in isolated simulations of each application function. A simulation run can yield the following results between two breakpoints: Total execution time, number of cache hits, number of cache misses, number of writes. By taking care that no other tasks are active in the system, these values can directly be attributed to one task. In our case study we use a benchmark input image for this purpose. This was sufficiently accurate as the nature of the algorithm is such that it shows no input data dependent behavior.

The cache offers single cycle access to the active thread, so that we consider the cache hit delay as part of the execution time. A cache miss will incur a waiting time for the requesting task that consists of the request latency via the bus plus the access time to the memory. Although the delays are actually input parameters to the simulator, we have independently determined them through measurements.

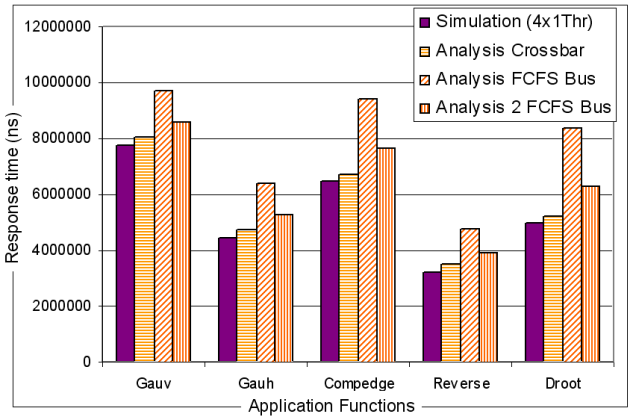


**Figure 7: Experiments for Singlethreaded Setup**

Figure 7 shows the results of the first experiment. Each of the 5 application functions is presented individually. The first bar represents simulated execution time if a dissected input image is concurrently processed by the four CPUs. Next, we performed our formal analysis with the data previously gathered from the exclusive function simulation (second bar). As there are no additional conflicts on processors, crossbar, or the memory, we receive very accurate results that closely resemble the simulation.

Now we modify the model of the bus and the memory to exclusively treat one request at a time in a first-come-first-served ordering. This is easily introduced into the analysis of each task by including the memory interference in the tasks' accumulated busy times of Equation 1. The conservative model of the interference will now contain all memory accesses by the tasks that are active at the same time. The third bar in Figure 7 shows the predicted response time for each application function. The response times of the functions are affected by the contention on the bus and memory to different degrees. Depending on the amount of memory traffic the response times increase by 25% for *Gauv* and up to 41% for *Droot*. In a final option we assume a hypothetical memory and bus controller that allows two parallel accesses which reduces the interference by half (4th bar). A designer can now choose the cheapest bus structure that is still guaranteed to deliver sufficient performance.

The second series of experiments assumes each application function is parallelised into 8 subtasks. Again, we derived single subtask behavior by simulation in isolation. The first two bars in Figure 8 show that our approach can again precisely capture the actual behavior for 8 concurrent subtasks on 8 cores.

We then assume that two subtasks are mapped to hardware threads on the same processor. This will cause competition for the processor, and also for the cache content. The third bar shows that for most functions (*Gauv*, *Gauh*,

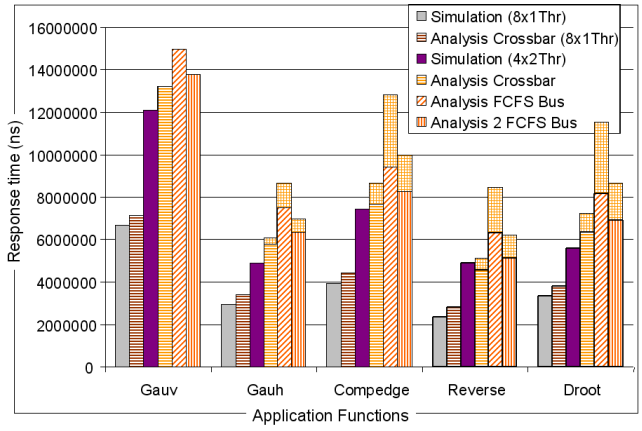


**Figure 8: Experiments for Multithreaded Setup**

*Compedge*, and *Droot*) processor sharing increases the measured response time by less than 100%. This can be attributed to how efficiently the memory accesses interleave during runtime. However, the measured response time for *Reverse* is more than twice as large: By mapping two tasks to the same core, the required execution time will remain unaffected, but the cache miss rate may increase due to cache thrashing.

This change in cache behavior can be avoided by relying on local cache partitioning or analytically bounded with formal task analysis such as [5]. Neither method is in place in our setup, so in order to account for this interference, we have measured the additional cache misses for each function observed under dualthreaded simulations. In general, simulation is unreliable to find worst-case cache misses due to the large space of possible application and cache states. In the given setup however, the state space is much smaller, because a) the input data does not impact the number of cache misses and b) the thread-offsets vary only insignificantly due to the fork-join structure of the application. The contribution of this effect to the response time is shown in the respective upper parts of each column.

Also for the setup with 4 dual-threaded processors, we explore the option of utilizing shared FCFS busses which allow only one or two simultaneous transactions. Functions which perform more memory accesses (*Compedge*, *Reverse*, or *Droot*) again suffer more severely from the resulting bus competition (as seen in the last two bars).

The overall analysis speed was very high. Each simulation run of *individual* task functions already took minutes to complete and had to be repeated several times, which becomes a severe problem if system level options are investigated. By contrast, each analysis result was calculated in less than a second due to the abstraction from the actual functionality.

## 5. CONCLUSION

In this paper a formal performance methodology and analysis has been applied to a realistic embedded multiprocessor system on chip. This was possible by addressing and quantifying the impact of the complex interdependencies that surface when shared memories are used. We capture the local task interaction in the multithreaded round-robin scheduler in our analysis allowing the prediction of the worst case response times. The memory accesses are analysed with unmatched speed and precision by relying on the concept of accumulated busy times instead of deriving individual request timing. We have used this approach to gather worst case performance metrics and quickly derive accurate estimates for various interconnect options.

## 6. REFERENCES

[1] R. Racu, R. Ernst, K. Richter, and M. Jersak. A Virtual Platform for Architecture Integration and Optimization in Automotive Comm. Networks. *SAE Congress*, 2007.

[2] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen. Predictable embedded multiprocessor system design. *In Proc. SCOPES workshop, Amsterdam*, 2004.

[3] P.G. Paulin, C. Pilkington, and E. Bensoudane. StepNP: a system-level exploration platform for network processors. *Design & Test of Computers, IEEE*, 19, 2002.

[4] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. *21st Intl. Conference on VLSI Design*, 2008.

[5] R. Wilhelm et al. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[6] J. Stohr, A. Bulow, and G. Farber. Bounding Worst-Case Access Times in Modern Multiprocessor Systems. *17th Euromicro Conference on Real-Time Systems*, 2005.

[7] Axel Siebenborn, Oliver Bringmann, and Wolfgang Rosenstiel. Worst-case performance analysis of parallel, communicating software processes. In *Intl. Symposium on Hardware/Software Codesign, Estes Park*, 2002.

[8] T. Henriksson, P. van der Wolf, A. Jantsch, and A. Bruce. Network Calculus Applied to Verification of Memory Access Performance in SoCs. *Workshop on Embedded Systems for Real-Time Multimedia*, 2007.

[9] S. Schliecker, M. Ivers, and R. Ernst. Integrated analysis of communicating tasks in MPSoCs. *Intl. Conference on Hardware/Software Codesign and System Synthesis*, 2006.

[10] P.G. Paulin, C. Pilkington, et al. Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2006.

[11] M. Paganini. Nomadik(TM): A Mobile Multimedia Application Processor Platform. *In Proc. ASP - Design Automation Conference*, 2007.

[12] Y. Bouchebaba, G. Nicolescu, E. Aboulhamid, and F. Coelho. Buffer and register allocation for memory space optimization. *In Proc. Intl. Conf. on Application-specific Systems, Architectures, and Processors*, 2006.

[13] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis-the SymTA/S approach. *Computers and Digital Techniques*, 152, 2005.

[14] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. *In Proc. Design Automation and Test in Europe*, 2003.

[15] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29, 1986.

[16] R. Racu, L. Li, R. Henia, A. Hamann, and R. Ernst. Improved response time analysis of tasks scheduled under preemptive Round-Robin. *In Proc. Intl. Conference on Hardware/Software Codesign and System Synthesis*, 2007.

[17] P. Crowley and J.L. Baer. Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors. *In Proc. 2nd Workshop on Network Processors*, 2003.

[18] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Reliable Performance Analysis of a Multicore Multithreaded System-On-Chip (with appendix). Technical Report 22837, Technische Universität Braunschweig, 2008.

# APPENDIX

## A. MULTITHREADED ROUND-ROBIN

In this section we introduce a scheduling analysis to determine the response time of tasks in the StepNP setup as presented in Section 2. First a worst case performance analysis of the scheduler for a set of simple tasks is presented, which we then extend to communicating tasks.

The utilized scheduler behaves as follows: A uniprocessor with $N$ hardware threads scheduled under the *Multithreaded Round-robin (MTRR) Scheduling* policy will cyclically offer $t_{slot}$ of execution time to each thread. Each thread that is not ready will be skipped with no additional overhead. As long as the number of tasks does not outgrow the number of hardware threads, all activations of a task $\tau_i$ are serviced by thread $i$ in order of their arrival. In the given system $t_{slot}$ is equivalent to the execution time of a single instruction.

By definition, a *simple task* $\tau_i$ becomes *ready* when an activating event has arrived. The maximum number of events in a time window of size $w$ that can lead to the activation of a task $i$ in a time is bounded by $\eta_i^+(w)$ (as provided in the iterative analysis of Sec. 3). An activation $i$ of a task $\tau_i$ is *finished* when it has executed on the processor for a time no longer than $C_i$ corresponding to its worst case core execution time. An activation of a simple task is continuously ready (but not necessarily executing) once it has been activated until it is finished.

To derive the worst case response time of such a task activation, we will first bound the delay that such an activation can experience due to other threads executing.

LEMMA 1. *Given hardware threads $i, j$, let thread $i$ be ready for $s_i$ consecutive time slots within an arbitrary time window of size $w_i$. Within this time window, thread $j$ may keep thread $i$ from executing for no more than*

$$d_j(w) \leq t_{slot} \cdot \min\{s_i, \left\lceil \frac{E_j(w)}{t_{slot}} \right\rceil\} \tag{2}$$

*where $E_j(w)$ is the maximum amount of execution requested by thread $j$ during $w$.*

PROOF. For each thread $j$ the amount of interference it may inflict on another thread (e.g. thread $i$) is bounded by both a) the workload to be processed by thread $j$ and b) the number of time slots offered to thread $j$. Both can be bounded by two simple observations:

*a) Thread Workload:* The maximal accumulated execution requirement for a thread $j$ which services task $\tau_j$ depends on the time window size $w$ and is bounded by the maximum number of activations of $\tau_j$ in this time window: $E_j(w) = \eta_j^+(w) \cdot C_j$. Thus, thread $j$ will have no remaining workload after it has received $\lceil E_j(w)/t_{slot} \rceil$ time slots, and will consequently waive any further time slots.

*b) Thread Execution Opportunities:* Let thread $i$ be ready for $s_i$ consecutive time slots. Then under *MTRR* every other thread is offered exactly $s_i - 1$ time slots in between. Furthermore, when the thread $i$ has become ready for the first time, every other thread is offered at most one time slot before $i$ receives its first time slot. Thus, $s_i$ consecutive time slots of the same thread can be delayed by at most $s_i$ time slots occupied per other thread.

Thus, the delay $d_j(w)$ that each thread $j$ can cause to the $s_i$ time slots of thread $i$ within a time window of size $w$ is bounded by Equation 2. □

THEOREM 1. *The response time of a task that requires $s_i$ consecutive time slots of thread $i$ is bounded by*

$$R_i \leq t_{slot} \cdot s_i + \sum_{j \in \{\mathbf{T} \setminus i\}} d_j(R_i) \tag{3}$$

*where $t_{slot}$ is the duration of one time slot, $\mathbf{T}$ are the utilized hardware threads, and $d_j(R_i)$ is the delay imposed by a hardware thread $j$ during the response time $R_i$.*

PROOF. The thread $i$ requires to be assigned $s_i = \lceil C_i/t_{slot} \rceil$ time slots to finish an activation of task $i$. The processor arbitrating the hardware threads $\mathbf{T}$ allows each other thread $\{j \in \mathbf{T} \setminus i\}$ to delay the execution of thread $i$ by at most $d_j(w)$ (see Lemma 1, and thus at most $d_j(R_i)$ within $i$'s response time $R_i$. Thus, the worst case response time of an activation of task $i$ fulfills Equation 3. □

Equation 3 shows $R_i$ on both sides and can not directly be solved. However, due to the monotonicity of the right hand side of Equation 3 it can be efficiently solved through iteration as proposed in [15]. Initially a small value for $R_i^0 = 0$ is assumed and the result is recursively applied until a fix point is reached.

## B. CONSIDERING SHARED MEMORY ACCESSES

The challenge in the given scheduling policy is to consider the effect of memory accesses during the execution of a task. These accesses will cause voluntary suspension of the tasks (task *waiting*), which will cause the corresponding thread to be skipped until the requested data is available.

A *Communicating Task* (as depicted Figure 5) behaves like a simple task, but additionally requires data from the shared memory during its execution. Let $Q_i$ be the set of requests $q_{i,n}$ with $0 \leq n < |Q_i|$ defining the target and address of the requests. These requests can be explicit memory accesses or implicit cache misses. The task is said to be *waiting*, when it has sent a request for data, but the data has not arrived at the processor. The response time of a communicating task is given by the sum of its ready times plus the time it is not ready, which is its waiting time.

The analysis requires a safe upper bound on the amount of memory requests. For a given task, this can be derived with static analysis approaches such as [5] for both explicit data fetch operation and implicit cache misses. Alternatively, as performed in Sec. 4, simulation of individual tasks can be used to gather adequate estimates.

The requests will separate the task into $|Q_i| + 1$ subsequences of time slots during which the task is continuously ready with arbitrary lengths $s_{i,0}, s_{i,2}, ...s_{i,|Q_i|}$. The sum of these subsequence lengths constitutes the original number of occupied time slots ( $\sum_{n=0..|Q_i|} s_{i,n} = s_i$). In the StepNP setup, memory request can only occur at the end of a time slot, because the time slots are assigned for single instructions only.

For every memory request $q_{i,n}$, task activation $i$ will be in the waiting state for a time $w_{i,n}$ while the data is requested over the bus, provided by the memory and transferred back to the CPU. Afterwards the requesting thread has to wait for the next time slot in order be serviced. It is then continuously ready for $s_{i,n+1}$ time slots before it requests the next data and becomes waiting again. Let this time be $R(s_{i,n+1})$.

Then, an activation is finished after:

$$R_i = (\sum_{n=0..|Q_i|} R(s_{i,n}) + w_{i,n}) + R(s_{i,|Q_i|+1}) \quad (4)$$

Recall Equation 2 delivers a conservative bound on the number of time slots occupied by a thread $j$, servicing simple tasks, before thread $i$ has received the opportunity to execute $s_i$ time slots. Communicating tasks with the same execution requirement $E_j(w)$ may not impose more interference in the same time frame. Thus Equations 2 and 3 can still be used to bound the response time of any subsequence $s_{i,n}$ of time slots of thread $i$ (i.e. $R(s_{i,n})$). It now remains to compute the intermediate waiting times $\sum w_{i,n}$.

The key idea is to consider all requests during the lifetime of a task jointly. We introduce the worst case *accumulated busy time*, which is defined as the total amount of time during which *at least one* request has been issued but is not finished. The accumulated busy time of the requests to the bus and the memory then exactly correspond to the sum of the task waiting times required in Equation 4 ($\sum w_{i,n}$).

This accumulated busy time can be efficiently calculated e.g. for a shared bus: A set of requests is issued from different processors that may interfere with each other. The exact individual request times are unknown and their actual latency ($w_{i,n}$) is highly dynamic. Extracting detailed timing information (e.g. when a specific cache miss occurs) is virtually impossible, and considering such details in a conservative analysis highly exponential. Consequently, we waive such details and focus on bounding the accumulated busy time ($\sum w_{i,n}$). It can straightforwardly be bounded by the sum of all involved transmission times, including those initiated from other processors. Given a certain dynamism in the system, this consideration will not result in excessive overestimations.

Without bus access prioritization, it has to be assumed that it is possible for every transaction issued by *any* processor during the lifetime of a task activation $i$ that they will disturb the transactions issued by $i$. In the present setup this is given by the requests issued by the other concurrently active tasks on the other processors, as well as the tasks on the same processor as their requests are treated first-come-first-served.

Thus, the accumulated busy time can be bounded as follows:

$$S(Q_i, w) \quad \leq \quad \sum_{p \in P} \sum_{\tau \in p} \sum_{q \in Q_\tau} C(q) + \sum_{\tau \in p_i} \sum_{q \in Q_\tau} C(q) \quad (5)$$

$Q_i$ are the requests sent by task $\tau_i$ that is mapped to processor $p_i$. $w$ is the time window within which these are sent.

$P$ is the set of processors in the system to which $\tau_i$ is not mapped.

$\tau$ is a task mapped to a processor $p$ and $Q_\tau$ is the set of requests to the shared resource issued by all activations of task $\tau$ in the time window $w$.

$C(q)$ is the maximum time that the request $q$ occupies the shared resource.

As opposed to Equation 1, Equation 5 delivers improved modeling opportunities: Requests by different tasks and sequences of requests from the same task can be differentiated and trigger a variable execution demand. Request prioritization can be easily introduced by annotating $C(q) = 0$ for all requests which may be preempted by higher priority requests. Non-preemptive sections (usually the length of a single memory request) can be captured with an additional blocking term.

This allows us to expand Equation 4 by calculating the local execution and ready times according to Equation 3 and the sum of the waiting times according to Equation 5:

$$R_i \quad \leq \quad S(Q_i, R_i) + t_{slot} \cdot (s_i + \sum_{j \in \{T_j \setminus i\}} d_j(R_i)) \quad (6)$$

This is an upper bound of the response time of a task $i$, including its accesses to a shared memory. As before, the right hand side of the Equation 6 is monotonic with respect to growing $R_i$, and can thus be solved iteratively until a fixed-point, the response time, has been found.